Laporan Tugas Besar 2

IF3170 Inteligensi Artifisial

Implementasi Algoritma Pembelajaran Mesin

Semester I Tahun 2024/2025

Disusun Oleh:

Dewantoro Triatmojo 13522011

Nyoman Ganadipa 13522066

Julian Chandra Sutadi 13522080

Rayhan Fadhlan Azka 13522095

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2024

Daftar Isi

Bab 1: Penjelasan Implementasi Model

1.1 Model KNN

Berikut merupakan penjelasan step by step untuk model KNN yang sudah dioptimasi dengan threading:

1. Inisialisasi Model KNN:
   Class diinisialisasi dengan menerima beberapa parameter pada *constructor* seperti `n_neighbors` untuk menentukan jumlah tetangga terdekat yang akan dipertimbangkan, `metric` untuk menentukan jenis perhitungan jarak yang akan digunakan (euclidean, manhattan, atau minkowski), parameter `p` untuk metric minkowski, `weights` untuk menentukan bobot voting (*uniform* atau *distance*), dan terakhir `n_jobs` untuk menentukan jumlah proses eksekusi paralel. Pada tahap ini juga terdapat validasi input untuk memastikan parameter yang diberikan valid dan sesuai dengan requirement. Storage untuk menyimpan data training (X), label (y) dan cache jarak juga diinisialisasi.

2. Validasi Data Proses:
   Validasi data dilakukan untuk memastikan format data yang akan diproses sesuai. Data input X dan label y dikonversi menjadi numpy array agar bisa diproses secara efisien. Dilakukan juga pengecekan konsistensi jumlah sampel antara X dan y untuk memastikan setiap sampel memiliki label. Jika jumlah sampel tidak sama, akan mengembalikan ValueError. Fungsi ini membantu mencegah error saat pemrosesan data di tahap selanjutnya.

3. Training Model:
   Pada tahap training, model menyimpan data training X dan label y yang sudah divalidasi. Kelas-kelas unik dari label y juga disimpan untuk digunakan dalam proses prediksi. Cache jarak dibersihkan setiap kali model di-fit ulang untuk menghindari penggunaan cache yang sudah tidak valid. Training KNN sebenarnya hanya menyimpan data, karena algoritma ini termasuk lazy learning yang baru melakukan komputasi saat prediksi.

4. Perhitungan Jarak
   Implementasi perhitungan jarak mendukung tiga metrik yaitu: *Manhattan distance*, *Euclidean distance*, dan *Minkowski distance*. Perhitungan dilakukan secara vectorized menggunakan numpy untuk efisiensi. Fungsi ini menghitung jarak antara satu titik query dengan seluruh titik training sekaligus, mengembalikan array jarak.

5. Prediksi Single Sample
   Untuk memprediksi satu sampel, pertama dilakukan pengecekan cache untuk menghindari perhitungan ulang jarak yang sudah pernah dihitung. Jika tidak ada di

cache, jarak dihitung dan disimpan. Kemudian diambil k tetangga terdekat berdasarkan jarak. Prediksi kelas dilakukan dengan voting - bisa dengan cara *uniform* (setiap tetangga memiliki bobot sama) atau *weighted by distance* (tetangga yang lebih dekat memiliki bobot lebih besar).

6. Prediksi Batch Samples
   Prediksi untuk multiple samples bisa dilakukan secara sekuensial atau paralel tergantung parameter `n_jobs`. Jika `n_jobs=1`, prediksi dilakukan satu per satu. Jika `n_jobs>1`, digunakan `ThreadPoolExecutor` untuk memproses beberapa prediksi secara paralel. Hal ini dapat mempercepat proses prediksi terutama untuk dataset yang sangat besar. Hasil akhirnya adalah array prediksi untuk semua sampel input.

Berikut source code dari Model KNN:

```
class OptimizedKNN:
      """
      Optimized K-Nearest Neighbors classifier with parallel processing
capabilities.

      Features:
      - Multiple distance metrics (Euclidean, Manhattan, Minkowski)
      - Parallel processing for faster predictions
      - Efficient distance calculations using vectorization
      - Distance caching for repeated predictions
      - Support for weighted voting

      Parameters:
      -----------
      n_neighbors : int, default=5
      Number of neighbors to use for prediction
      metric : str, default="euclidean"
      Distance metric to use. Options: "euclidean", "manhattan", "minkowski"
      p : int, optional
      Power parameter for Minkowski metric
      weights : str, default="uniform"
      Weight function used in prediction. Options: "uniform", "distance"
      n_jobs : int, default=1
      Number of parallel jobs. If -1, use all available CPU cores
      """

      def __init__(self, n_neighbors=5, metric="euclidean", p=None,
                   weights="uniform", n_jobs=1):

      if not isinstance(n_neighbors, int) or n_neighbors <= 0:
            raise ValueError(f"n_neighbors must be a positive integer, got
{n_neighbors}")

      if metric not in ["euclidean", "manhattan", "minkowski"]:
```

```
            raise ValueError(f"Unsupported metric: {metric}")

        if weights not in ["uniform", "distance"]:
            raise ValueError(f"Unsupported weight function: {weights}")

        if metric == "manhattan" and p not in (None, 1):
            raise ValueError("When metric='manhattan', p should be None or 1")

        if metric == "euclidean" and p not in (None, 2):
            raise ValueError("When metric='euclidean', p should be None or 2")

        if metric == "minkowski" and p is None:
            raise ValueError("For metric='minkowski', p parameter is required")

        self.n_neighbors = n_neighbors
        self.metric = metric
        self.p = p
        self.weights = weights
        self.n_jobs = n_jobs if n_jobs != -1 else None


        self.X = None
        self.y = None
        self._distance_cache = {}

        def _validate_data(self, X, y=None):
        """Validate and convert input data to numpy arrays."""
        X = np.asarray(X)
        if y is not None:
            y = np.asarray(y)
            if len(X) != len(y):
            raise ValueError("X and y must have the same number of samples")
        return X, y

        def fit(self, X, y):
        """
        Fit the KNN model.

        Parameters:
        -----------
        X : array-like of shape (n_samples, n_features)
            Training data
        y : array-like of shape (n_samples,)
            Target values

        Returns:
        --------
        self : OptimizedKNN
            The fitted model
```

```
      """
      self.X, self.y = self._validate_data(X, y)
      self._classes = np.unique(y)
      self._distance_cache.clear()
      return self

      def _compute_distances(self, x):
      """Compute distances between a single point and all training points."""
      if self.metric == "manhattan":
            return np.sum(np.abs(self.X - x), axis=1)
      elif self.metric == "euclidean":

            return np.sqrt(np.sum((self.X - x) ** 2, axis=1))
      else:
            return np.power(np.sum(np.power(np.abs(self.X - x), self.p),
axis=1), 1/self.p)

      def _predict_single(self, x):
      """Predict class for a single sample."""
      # Check cache first
      x_key = hash(x.tobytes())
      if x_key in self._distance_cache:
            distances = self._distance_cache[x_key]
      else:
            distances = self._compute_distances(x)
            self._distance_cache[x_key] = distances

      # Get nearest neighbors
      nearest_indices = np.argpartition(distances,
self.n_neighbors)[:self.n_neighbors]
      nearest_distances = distances[nearest_indices]
      nearest_labels = self.y[nearest_indices]

      if self.weights == "uniform":
            # Simple majority voting
            return Counter(nearest_labels).most_common(1)[0][0]
      else:  # distance weighting
            # Avoid division by zero
            weights = 1 / (nearest_distances + np.finfo(float).eps)
            weighted_votes = {}
            for label, weight in zip(nearest_labels, weights):
            weighted_votes[label] = weighted_votes.get(label, 0) + weight
            return max(weighted_votes.items(), key=lambda x: x[1])[0]

      def predict(self, X):
      """
      Predict class labels for samples in X.

      Parameters:
```

```
        -----------
        X : array-like of shape (n_samples, n_features)
              Samples to predict

        Returns:
        --------
        y_pred : array of shape (n_samples,)
              Predicted class labels
        """
        X, _ = self._validate_data(X)

        if self.n_jobs == 1:
              return np.array([self._predict_single(x) for x in X])

        # Parallel prediction
        with ThreadPoolExecutor(max_workers=self.n_jobs) as executor:
              predictions = list(executor.map(self._predict_single, X))
        return np.array(predictions)

        def get_params(self):
        """Get model parameters."""
        return {
              'n_neighbors': self.n_neighbors,
              'metric': self.metric,
              'p': self.p,
              'weights': self.weights,
              'n_jobs': self.n_jobs
        }
```

1.2 Model Naive-Bayes

Berikut merupakan penjelasan step by step untuk model Naive Bayes:

1. Inisialisasi Model Naive Bayes:
   Implementasi dari algoritma Naive Bayes dengan asumsi data memiliki distribusi Gaussian (normal) terhadap variabel target. Pada tahap inisialisasi, tidak ada parameter yang perlu diatur karena semua parameter model akan dihitung saat proses fitting atau pelatihan data.

2. Proses Fitting Model
   Pada method fit, kode melakukan perhitungan penting untuk membangun model. Pertama, kode menghitung jumlah kelas unik dan frekuensi kemunculannya dari data target (y). Kemudian, hitung probabilitas prior untuk setiap kelas dengan membagi jumlah sampel per kelas dengan total sampel. Selanjutnya, untuk setiap kelas dan setiap fitur, dihitung nilai mean (theta) menggunakan Maximum Likelihood Estimation dan variance menggunakan Minimum Variance Unbiased Estimation. Ditambahkan juga

nilai kecil (1e-9) pada variance untuk mencegah pembagian dengan nol yang bisa menyebabkan error.

3. Implementasi Prediksi
   Dalam method predict, kode mengimplementasikan perhitungan probabilitas untuk melakukan klasifikasi. Proses dimulai dengan mengkonversi probabilitas prior ke dalam bentuk logaritmik untuk mencegah underflow numerik. Kemudian, untuk setiap kelas, hitung log likelihood menggunakan fungsi densitas probabilitas Gaussian. Perhitungan ini menggunakan rumus PDF Gaussian yang telah dikonversi ke bentuk logaritmik untuk stabilitas numerik yang lebih baik.

4. Tahap Prediksi Akhir
   Pada tahap akhir prediksi, kode mengkonversi log likelihood yang telah dihitung menjadi array numpy dan melakukan transpose untuk mendapatkan format yang sesuai. Selanjutnya, log posterior dihitung dengan menambahkan log prior dengan log likelihood. Terakhir, kode mengembalikan prediksi kelas dengan mengambil argmax dari log posterior, yang memberikan indeks kelas dengan probabilitas tertinggi untuk setiap sampel.

Berikut source code dari Model Naive-Bayes:

```
import numpy as np

class GaussianNB():
      """
      Ref: https://www.cs.cmu.edu/~tom/mlbook/NBayesLogReg.pdf
      Uses estimate MLE for mean and MVUE for variance
      """

      def __init__(self) :
      pass

      def fit(self, X, y):
      self.classes_, class_count_ = np.unique(y, return_counts=True)

      # ndarray of shape (n_classes,)
      self.class_priors_ = class_count_ / y.shape[0]

      # ndarray of shape (n_classes, n_features)
      self.theta_ = np.array([X[y == c].mean(axis=0) for c in self.classes_])
      self.var_ = np.array([X[y == c].var(axis=0) for c in self.classes_])

      self.var_ = np.maximum(self.var_, 1e-9)

      def predict(self, X):
      log_priors = np.log(self.class_priors_)

      log_likelihoods = []
```

```
      for i, _ in enumerate(self.classes_):
            # likelihood: feature xi given class Ck
            # X - self.theta_[i] involves broadcasting self.theta_[i] to each
row of X
            log_likelihood = -0.5 * np.sum(np.log(2 * np.pi * self.var_[i])) -
0.5 * np.sum(((X - self.theta_[i]) ** 2) / self.var_[i], axis=1)
            log_likelihoods.append(log_likelihood)

      log_likelihoods = np.array(log_likelihoods).T
      log_posteriors = log_likelihoods + log_priors

      return self.classes_[np.argmax(log_posteriors, axis=1)]
```

1.3 Model ID3

1. Struktur Node dan Representasi Pohon
   Dalam implementasi algoritma ID3, struktur data utama yang digunakan adalah pohon keputusan yang terdiri dari node-node yang saling terhubung. Setiap node dalam pohon berfungsi sebagai node keputusan atau node daun. Node keputusan berisi informasi tentang fitur yang digunakan untuk pemecahan (splitting), serta reference ke node anak kiri dan kanan yang merupakan hasil dari pemecahan. Sementara itu, node daun menyimpan nilai klasifikasi akhir yang akan digunakan untuk prediksi.

2. Komponen Utama Algoritma ID3
   Algoritma ID3 memiliki beberapa komponen yang saling bekerja sama untuk membangun pohon keputusan yang optimal. Pertama, algoritma menggunakan konsep

   entropy, yang dihitung menggunakan rumus $-\sum(p_i \, log_2(p_i))$ dimana p_i adalah

   probabilitas kelas i. Kedua, information gain digunakan, yang dihitung sebagai selisih antara entropy parent node dan rata-rata tertimbang entropy child nodes. Ketiga, proses pemilihan fitur dilakukan dengan mengevaluasi setiap kemungkinan pemecahan untuk fitur numerik dan kategorikal, dengan tujuan menemukan pemecahan yang menghasilkan information gain maksimal.

3. Proses Konstruksi Pohon Keputusan
   Pembangunan pohon keputusan dalam algoritma ID3 dilakukan secara rekursif menggunakan pendekatan DFS, dimulai dari root node hingga leaf nodes. Pada setiap tahap konstruksi, algoritma menghitung information gain untuk semua kemungkinan pemecahan fitur dan memilih fitur dengan information gain tertinggi sebagai kriteria pemecahan. Setelah pemilihan fitur, algoritma membuat node anak berdasarkan pemecahan tersebut dan melanjutkan proses pembangunan secara rekursif untuk setiap node anak. Proses ini berlanjut hingga mencapai salah satu kriteria penghentian, seperti mencapai kedalaman maksimum, jumlah sampel minimum, atau ketika nilai entropi sudah 0 (semua sampel dalam node termasuk dalam kelas yang sama).

5. Penanganan Berbagai Tipe Fitur
   Implementasi algoritma ID3 ini dirancang untuk dapat menangani baik fitur numerik maupun kategorikal secara efektif. Untuk fitur numerik, algoritma menggunakan pendekatan pemecahan biner berbasis nilai ambang batas, di mana data dipisahkan menjadi dua kelompok berdasarkan apakah nilainya lebih kecil atau lebih besar dari ambang batas. Sementara untuk fitur kategorikal, pemecahan dilakukan berdasarkan perbandingan kesamaan nilai. Fleksibilitas ini memungkinkan algoritma untuk diterapkan pada berbagai jenis dataset dengan karakteristik fitur yang berbeda.

6. Proses Prediksi
   Setelah pohon keputusan terbentuk, proses prediksi dilakukan dengan cara melintasi pohon dari root node hingga mencapai leaf node. Pada setiap node internal, algoritma mengevaluasi kondisi pemecahan berdasarkan nilai fitur input dan menentukan apakah harus bergerak ke node anak kiri atau kanan. Proses ini berlanjut hingga mencapai leaf node, di mana nilai klasifikasi yang tersimpan di node tersebut akan dijadikan sebagai hasil prediksi.

7. Kriteria Penghentian
   Algoritma ID3 menggunakan beberapa kriteria penghentian untuk mengontrol pertumbuhan pohon dan mencegah overfitting. Kriteria-kriteria ini mencakup batas kedalaman maksimum pohon, jumlah sampel minimum yang diperlukan untuk melakukan pemecahan pada sebuah node, dan tingkat kemurnian node. Ketika salah satu dari kriteria ini terpenuhi, proses pembangunan pohon pada cabang tersebut akan dihentikan dan node tersebut akan dijadikan leaf node.

Berikut source code ID3 Model:

```
import numpy as np
from collections import Counter
import pandas as pd
from concurrent.futures import ThreadPoolExecutor
from typing import List, Optional, Union, Tuple
import threading

class Node:
      def __init__(self, feature=None, threshold=None, left=None, right=None,
value=None):
      self.feature = feature
      self.threshold = threshold
      self.left = left
      self.right = right
      self.value = value

class ID3DecisionTree:
      def __init__(self, max_depth: Optional[int] = None,
                   min_samples_split: int = 2,
```

```
                 n_jobs: int = -1):
        """
        Initialize OptimizedID3DecisionTree classifier

        Parameters:
        -----------
        max_depth : int, optional
              Maximum depth of the tree
        min_samples_split : int
              Minimum samples required to split a node
        n_jobs : int
              Number of parallel jobs. -1 means using all processors
        """
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.n_jobs = n_jobs if n_jobs > 0 else threading.active_count()
        self.root = None
        self.n_classes = None
        self.feature_types = None
        self.feature_names = None
        self._lock = threading.Lock()

        def fit(self, X: np.ndarray, y: np.ndarray,
              feature_types: Optional[List[str]] = None,
              feature_names: Optional[List[str]] = None) -> 'ID3DecisionTree':
        """
        Train the decision tree using parallel processing

        Parameters:
        -----------
        X : array-like of shape (n_samples, n_features)
              Training data
        y : array-like of shape (n_samples,)
              Target values
        feature_types : list of str, optional
              List of feature types ('numerical' or 'categorical')
        feature_names : list of str, optional
              List of feature names
        """
        X = np.array(X)
        y = np.array(y)

        self.n_classes = len(np.unique(y))
        self.feature_types = feature_types or ['numerical'] * X.shape[1]
        self.feature_names = feature_names or [f'feature_{i}' for i in
range(X.shape[1])]

        self.root = self._grow_tree(X, y)
        return self
```

```
      def _entropy(self, y: np.ndarray) -> float:
      """Calculate entropy of a node using vectorized operations"""
      hist = np.bincount(y)
      ps = hist / len(y)
      ps = ps[ps > 0]
      return -np.sum(ps * np.log2(ps))

      def _information_gain(self, y: np.ndarray, y_left: np.ndarray,
                        y_right: np.ndarray) -> float:
      """Calculate information gain for a split using vectorized operations"""
      if len(y_left) == 0 or len(y_right) == 0:
            return 0

      n = len(y)
      parent_entropy = self._entropy(y)
      n_l, n_r = len(y_left), len(y_right)
      child_entropy = (n_l / n) * self._entropy(y_left) + (n_r / n) *
self._entropy(y_right)
      return parent_entropy - child_entropy

      def _find_best_split_for_feature(self, X: np.ndarray, y: np.ndarray,
                                       feature_idx: int) -> Tuple[float,
Optional[Union[float, str]]]:
      """Find the best split for a single feature"""
      best_gain = -1
      best_threshold = None

      if self.feature_types[feature_idx] == 'numerical':
            thresholds = np.unique(X[:, feature_idx])

            for threshold in thresholds:
            left_mask = X[:, feature_idx] <= threshold
            if not (np.any(left_mask) and np.any(~left_mask)):
                  continue

            gain = self._information_gain(y, y[left_mask], y[~left_mask])
            if gain > best_gain:
                  best_gain = gain
                  best_threshold = threshold
      else:
            unique_values = np.unique(X[:, feature_idx])
            for value in unique_values:
            left_mask = X[:, feature_idx] == value
            if not (np.any(left_mask) and np.any(~left_mask)):
                  continue

            gain = self._information_gain(y, y[left_mask], y[~left_mask])
            if gain > best_gain:
```

```
                  best_gain = gain
                  best_threshold = value

      return best_gain, best_threshold

      def _parallel_find_best_split(self, X: np.ndarray, y: np.ndarray,
                                    feature_indices: List[int]) -> List[Tuple[int,
float, Optional[Union[float, str]]]]:
      """Find best splits for multiple features in parallel"""
      results = []
      for idx in feature_indices:
            gain, threshold = self._find_best_split_for_feature(X, y, idx)
            results.append((idx, gain, threshold))
      return results

      def _grow_tree(self, X: np.ndarray, y: np.ndarray, depth: int = 0) ->
Node:
      """Recursively grow the decision tree using parallel processing for
feature selection"""
      n_samples, n_features = X.shape

      # Check stopping criteria
      if (self.max_depth is not None and depth >= self.max_depth) or \
            n_samples < self.min_samples_split or \
            len(np.unique(y)) == 1:
            return Node(value=Counter(y).most_common(1)[0][0])

      # Split features into chunks for parallel processing
      feature_chunks = np.array_split(range(n_features), self.n_jobs)

      # Find best split using parallel processing
      best_gain = -1
      best_feature = None
      best_threshold = None

      with ThreadPoolExecutor(max_workers=self.n_jobs) as executor:
            future_results = [
            executor.submit(self._parallel_find_best_split, X, y, chunk)
            for chunk in feature_chunks if len(chunk) > 0
            ]

            for future in future_results:
            results = future.result()
            for feature_idx, gain, threshold in results:
                  if gain > best_gain:
                        best_gain = gain
                        best_feature = feature_idx
                        best_threshold = threshold
```

```
    # If no good split is found, create leaf node
    if best_gain == -1:
          return Node(value=Counter(y).most_common(1)[0][0])

    # Create split node
    if self.feature_types[best_feature] == 'numerical':
          left_mask = X[:, best_feature] <= best_threshold
    else:
          left_mask = X[:, best_feature] == best_threshold

    right_mask = ~left_mask

    # Recursively grow subtrees
    left_subtree = self._grow_tree(X[left_mask], y[left_mask], depth + 1)
    right_subtree = self._grow_tree(X[right_mask], y[right_mask], depth + 1)

    return Node(
          feature=best_feature,
          threshold=best_threshold,
          left=left_subtree,
          right=right_subtree
    )

    def predict(self, X: np.ndarray) -> np.ndarray:
    """Predict class for X using parallel processing"""
    X = np.array(X)

    def predict_batch(batch):
          return np.array([self._traverse_tree(x, self.root) for x in batch])

    # Split data into batches for parallel prediction
    batch_size = max(1, len(X) // self.n_jobs)
    batches = [X[i:i + batch_size] for i in range(0, len(X), batch_size)]

    with ThreadPoolExecutor(max_workers=self.n_jobs) as executor:
          predictions = list(executor.map(predict_batch, batches))

    return np.concatenate(predictions)

    def _traverse_tree(self, x: np.ndarray, node: Node) -> int:
    """Helper method to traverse the tree"""
    if node.value is not None:
          return node.value

    if self.feature_types[node.feature] == 'numerical':
          if x[node.feature] <= node.threshold:
          return self._traverse_tree(x, node.left)
          return self._traverse_tree(x, node.right)
    else:
```

```
            if x[node.feature] == node.threshold:
            return self._traverse_tree(x, node.left)
            return self._traverse_tree(x, node.right)
```

Bab 2: Penjelasan Tahap *Cleaning* dan *Preprocessing*

2.1 Tahap *Cleaning*

2.1.1 *Data Validation*

Pada proses data validation, kami menemukan bahwa is_ftp_login memiliki nilai yang bukan bernilai boolean (bukan 1 ataupun 0). Padahal kolom tersebut diekspektasikan untuk memiliki nilai binary. Di sini kami menghandle data yang diluar nilai binary tersebut untuk menjadi nilai modus dari kolom tersebut.

2.1.2 *Handling Missing Data*

Pemilihan metode *imputer* yang tepat sangat penting karena dapat mempengaruhi kualitas dan akurasi model yang akan dibuat. Untuk dataset *network traffic* yang sedang dianalisis, kami menggunakan pendekatan yang berbeda untuk *categorical features* dan *numerical features* berdasarkan karakteristik masing-masing tipe data.

Untuk *categorical features*: proto, service, state, is_sm_ips_ports, dan is_ftp_login, kami memilih untuk menggunakan *imputer* modus atau *most frequent*. Metode ini dipilih cukup *reasonable* untuk mengisi nilai yang hilang dengan nilai yang paling sering muncul dalam data. Metode ini membantu mempertahankan distribusi kategori yang ada dan merupakan pendekatan yang umum digunakan untuk data kategorikal.

Di sisi lain, untuk *numerical features*, kami menggunakan *imputer* median. Metode ini didasarkan pada hasil *exploratory data analysis* yang menunjukkan bahwa hampir seluruh fitur numerikal memiliki distribusi *right skewed*. Dalam kasus distribusi yang right skewed, median merupakan pilihan yang lebih *robust* dibandingkan mean karena tidak sensitif terhadap outlier atau nilai-nilai yang ekstrim nilainya. Sebagai contoh, dalam data lalu lintas jaringan, mungkin ada beberapa transaksi dengan ukuran byte yang sangat besar (seperti transfer file besar atau potensi serangan DDoS) yang bisa sangat mempengaruhi nilai mean, sementara median tetap memberikan estimasi yang stabil dari nilai tengah data.

Penggunaan *imputer* yang berbeda-beda ini mempertimbangkan karakteristik khusus dari setiap tipe fitur. Pendekatan ini sangat penting dalam konteks analisis lalu lintas jaringan, dimana akurasi dan keandalan data sangat penting untuk mendeteksi potensi serangan. Dengan menggunakan strategi imputasi yang sesuai, kami dapat memastikan bahwa dataset yang dihasilkan tetap mempertahankan karakteristik statistik yang penting dari data asli, tetapi juga mengatasi masalah *missing values* yang dapat menghambat proses analisis selanjutnya.

2.1.3. *Handling Outlier Data*

Proses penanganan outlier difokuskan hanya pada *numerical features*. Metode yang dipilih adalah kombinasi dari deteksi outlier menggunakan metode IQR (Interquartile Range) dan penanganan outlier menggunakan teknik clipping. Metode IQR dipilih untuk mendeteksi outlier karena kehandalannya dalam mengidentifikasi nilai-nilai yang secara statistik berada jauh dari sebagian besar data. IQR dihitung dengan mencari selisih antara kuartil ketiga (Q3) dan kuartil pertama (Q1) dari data. Sebuah entry dianggap sebagai outlier jika nilainya berada di luar rentang [Q1 - 1.5IQR, Q3 + 1.5IQR]. Pendekatan ini sangat efektif untuk data lalu lintas jaringan yang seringkali memiliki distribusi yang tidak normal dan mengandung nilai-nilai ekstrim. Saat suatu *entry* terdeteksi *outlier,* entry tersebut direplace menggunakan nilai *upper bound* atau *lower bound* yang batas IQR.

Metode ini dipilih karena: 1) metode clip memungkinkan kita untuk mempertahankan jumlah data yang ada sambil mengurangi pengaruh nilai-nilai yang sangat ekstrim yang bisa mengganggu analisis. 2) dalam kasus lalu lintas jaringan, outlier mungkin merepresentasikan kejadian yang benar-benar terjadi (seperti transfer file besar atau lonjakan traffic) yang tidak ingin kita hilangkan sepenuhnya dari dataset. Ketiga, dengan menggunakan clip, kita tetap mempertahankan informasi bahwa entry tersebut merupakan nilai ekstrem, yang bisa jadi merupakan informasi yang berguna untuk analisis keamanan jaringan.

2.1.4. *Remove duplicates*

Untuk handling duplicates, digunakan column ['proto', 'service', 'state', 'dur', 'sbytes', 'dbytes'] sebagai kumpulan kolom yang diperhatikan untuk melihat apakah suatu row mirip dengan row yang lain. Apabila entry pada pada kolom-kolom tersebut bernilai sama untuk beberapa row yang berbeda, maka row yang dipilih hanya row yang muncul paling awal, sisanya akan dibuang.

Metode ini dinilai sangat berguna untuk data *Network Traffic*, di mana duplikasi data bisa terjadi karena berbagai alasan seperti retransmisi paket. Dengan menghapus duplikat ini secara bertanggung jawab, kita dapat meningkatkan kualitas dataset untuk analisis lebih lanjut sambil mempertahankan transparansi dalam proses pembersihan data.

2.1.5. *Feature Engineering*

Feature Engineering yang dilakukan digunakan dibagi menjadi 4 proses:

1. *Feature Selection*, yang bertujuan untuk mengurangi dimensi data dengan menghilangkan fitur-fitur yang kurang informatif atau redundan. Proses ini dilakukan melalui dua tahap: menghilangkan fitur dengan varians rendah menggunakan

VarianceThreshold dan mengeliminasi fitur yang memiliki korelasi tinggi dengan fitur lainnya berdasarkan threshold yang ditentukan. Pendekatan ini membantu meningkatkan efisiensi model dengan mempertahankan hanya fitur-fitur yang paling relevan.

2. *Interaction Features*, yang menciptakan fitur-fitur baru berdasarkan hubungan antar fitur yang ada. Beberapa contoh interaction features yang dibuat termasuk *bytes per packet ratios* (yang menghitung rasio bytes terhadap packet untuk source dan destination), *load ratios* (yang membandingkan beban antara source dan destination), *loss ratios* (yang mengukur tingkat packet loss), dan *window ratios* (yang membandingkan ukuran window). Fitur-fitur interaksi ini membantu model dalam menangkap pola-pola kompleks dalam data lalu lintas jaringan.

3. *Binning Features*, di mana *numerical features* dibagi menjadi sejumlah bin dengan lebar yang sama. Teknik binning ini berguna untuk beberapa tujuan: menangani outlier dalam data, mengubah fitur numerical yang *continuous* menjadi bentuk kategorikal, dan meningkatkan ketahanan model terhadap noise dalam data. Jumlah bin dapat dikonfigurasi sesuai kebutuhan melalui parameter n_bins.

4. *Domain-Specific Features*, memanfaatkan pengetahuan khusus tentang domain lalu lintas jaringan. Fitur-fitur ini mencakup beberapa kategori: TCP connection features yang menghitung waktu total koneksi TCP, traffic intensity features yang mengukur total bytes dan packets, session features yang mencakup informasi tentang panjang sesi dan rata-rata ukuran paket, serta rate features yang menghitung *packet rate* dan *byte rate*. Fitur-fitur *domain-specific* ini membantu model dalam memahami karakteristik khusus dari lalu lintas jaringan.

2.2. Tahap *Preprocessing*

2.2.1. *Feature Scaling*

Kami melakukan uji dengan berbagai feature scaling, diantaranya dengan menggunakan StandardScaler, RobustScaler, dan MinMaxScaler. Pada akhirnya kami memutuskan untuk menggunakan StandardScaler karena menghasilkan model terbaik dibandingkan scaler lainnya. StandardScaler bekerja dengan cara mengubah data sehingga memiliki rata-rata (mean) 0 dan standar deviasi 1. Dalam kasus ini kami menggunakan feature scaling pada data numerical.

2.2.2. *Feature Encoding*

Feature Encoding bertujuan untuk mengubah data *categorical* menjadi bentuk numerik yang dapat diproses oleh model machine learning. Pada dataset *network traffic* ini, teknik encoding yang digunakan adalah One-Hot Encoding melalui OneHotEncoder dari scikit-learn.

One-Hot Encoding dipilih karena sangat cocok untuk fitur kategorikal seperti proto, service, dan state yang memiliki kategori nominal, dimana tidak ada urutan atau tingkatan antar kategori. Keuntungan menggunakan One-Hot Encoding dalam data *Network Traffic* adalah kemampuannya untuk mempertahankan informasi kategorikal tanpa menambahkan tingkatan di antaranya. Misalnya, tidak bisa dikatakan bahwa protokol TCP lebih besar atau lebih kecil dari UDP.

2.2.3. Preprocessing Lainnya

Beberapa tahapan data preprocessing yang telah disiapkan (*dimensionality reduction*, normalization, under/oversampling) tidak kami gunakan karena tidak membantu dalam meraih performansi model yang lebih baik. Misalnya, tanpa dimensionality reduction, kami telah mengurangi lebih dari setengah rows model yang dilatih menggunakan *domain knowledge*. Selain itu, karena model paling baik yang diperoleh adalah model berbasis tree (ID3), kami menganggap normalisasi ataupun standarisasi mungkin tidak diperlukan.

Bab 3: Perbandingan Hasil Prediksi

3.1. KNN

Implementasi Sendiri

	precision	recall	f1-score	support
0	0.53	0.25	0.34	144
1	0.48	0.11	0.18	111
2	0.32	0.17	0.22	802
3	0.74	0.83	0.78	4047
4	0.65	0.68	0.66	2644
5	0.92	0.53	0.68	238
6	0.91	0.89	0.90	8542
7	0.58	0.73	0.65	1014
8	0.41	0.24	0.31	229
9	1.00	0.04	0.07	27
accuracy			0.78	17798
macro avg	0.65	0.45	0.48	17798
weighted avg	0.77	0.78	0.77	17798

Accuracy: 0.7811551859759523

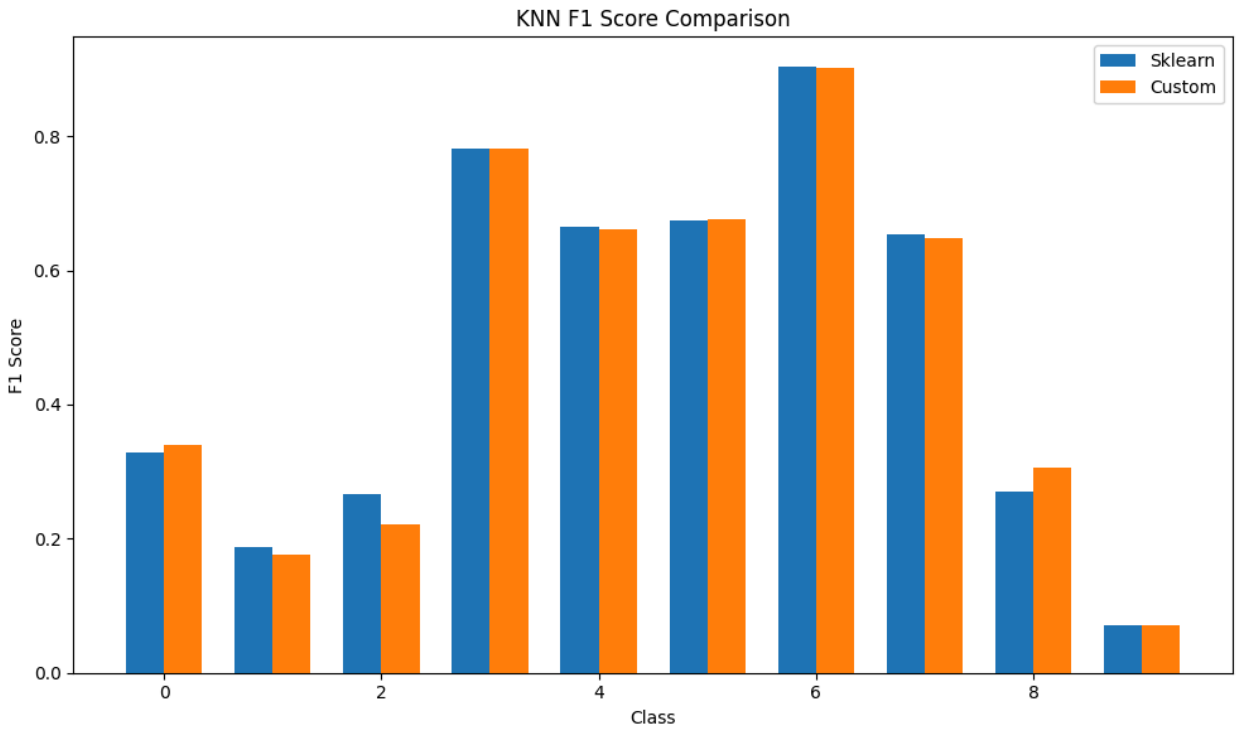
Menggunakan Pustaka

	precision	recall	f1-score	support
0	0.46	0.26	0.33	144
1	0.36	0.13	0.19	111
2	0.33	0.22	0.27	802
3	0.73	0.84	0.78	4047
4	0.65	0.69	0.67	2644
5	0.93	0.53	0.67	238
6	0.92	0.89	0.90	8542
7	0.61	0.71	0.65	1014
8	0.43	0.20	0.27	229
9	1.00	0.04	0.07	27
accuracy			0.78	17798
macro avg	0.64	0.45	0.48	17798
weighted avg	0.78	0.78	0.78	17798

Accuracy: 0.7819979773008203

Hasil prediksi menggunakan KNN yang diimplementasikan secara mandiri dengan yang menggunakan pustaka scikit-learn tidak jauh berbeda. Hal ini mungkin disebabkan tidak banyak parameter yang dapat digunakan untuk mengkustomisasi pengimplementasian algoritma KNN.

Untuk memastikan efisiensi, model kami mencoba agar semua operasi dapat dilakukan dalam operasi vektor, alih-aling menggunakan *loop*. Walau demikian, proses prediksi menggunakan model yang diimplementasikan sendiri lebih lama. Oleh karena itu, model kami memanfaatkan threading untuk mempercepat perhitungan.



3.2. Naive-Bayes

Implementasi Sendiri

	precision	recall	f1-score	support
0	0.09	0.65	0.15	144
1	0.03	0.23	0.05	111
2	1.00	0.00	0.00	802
3	0.82	0.42	0.55	4047
4	0.53	0.13	0.21	2644
5	0.12	0.59	0.20	238
6	0.99	0.69	0.81	8542
7	0.06	0.03	0.04	1014
8	0.05	0.99	0.09	229
9	0.01	0.15	0.01	27
accuracy			0.47	17798
macro avg	0.37	0.39	0.21	17798
weighted avg	0.79	0.47	0.55	17798

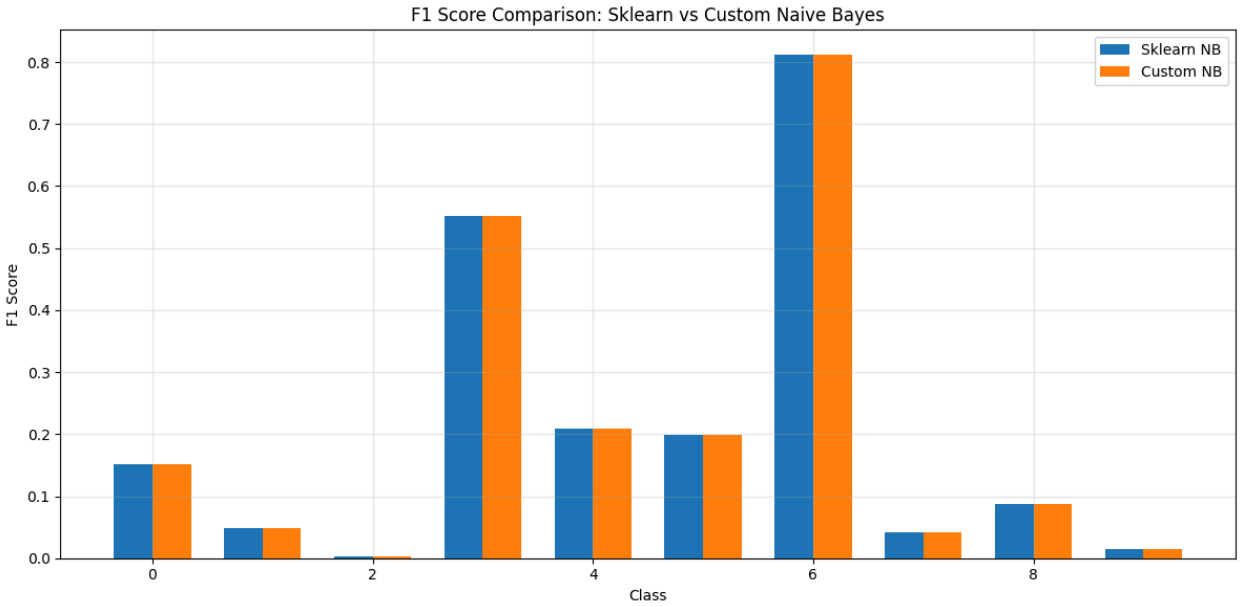
Accuracy: 0.47280593325092707

Menggunakan Pustaka

	precision	recall	f1-score	support
0	0.09	0.65	0.15	144
1	0.03	0.23	0.05	111
2	1.00	0.00	0.00	802
3	0.82	0.42	0.55	4047
4	0.53	0.13	0.21	2644
5	0.12	0.59	0.20	238
6	0.99	0.69	0.81	8542
7	0.06	0.03	0.04	1014
8	0.05	0.99	0.09	229
9	0.01	0.15	0.01	27
accuracy			0.47	17798
macro avg	0.37	0.39	0.21	17798
weighted avg	0.79	0.47	0.55	17798

Accuracy: 0.47280593325092707

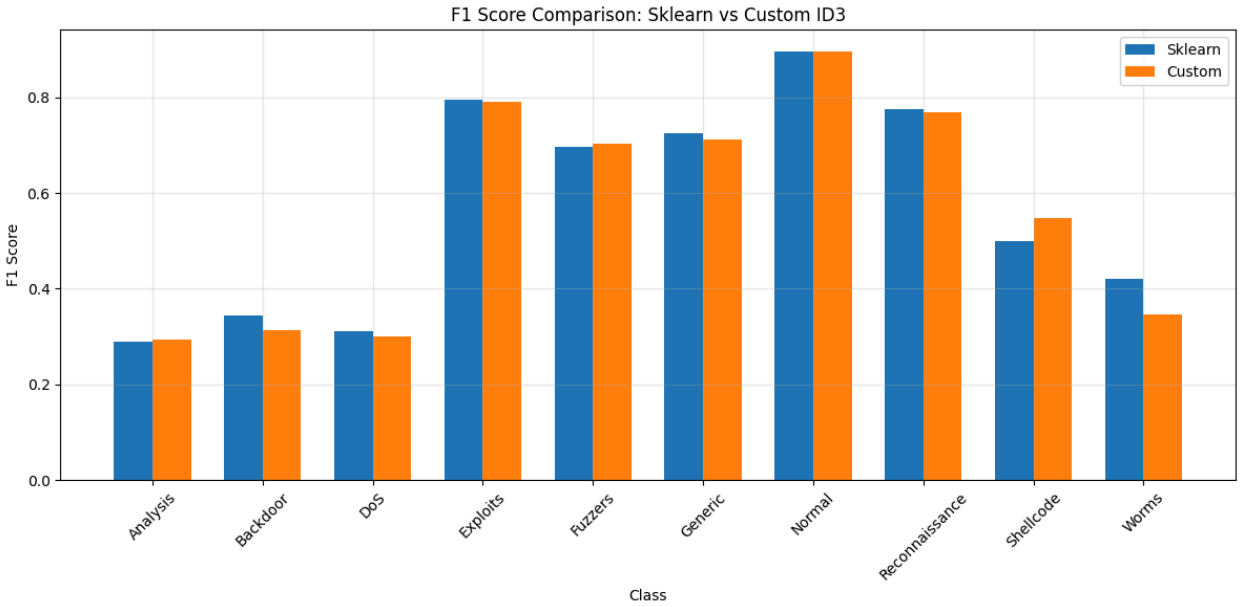
Model Naive-Bayes adalah model yang berdasarkan formulasi statistika matematik, sehingga proses pelatihannya relatif lebih cepat dan hasilnya sangat serupa dengan model bawaan pustaka.



3.3 ID3

Implementasi Sendiri				
	precision	recall	f1-score	support
0	0.46	0.22	0.29	144
1	0.51	0.23	0.31	111
2	0.37	0.25	0.30	802
3	0.77	0.82	0.79	4047
4	0.62	0.81	0.70	2644
5	0.74	0.69	0.71	238
6	0.93	0.86	0.90	8542
7	0.77	0.76	0.77	1014
8	0.56	0.53	0.55	229
9	0.36	0.33	0.35	27
accuracy			0.79	17798
macro avg	0.61	0.55	0.57	17798
weighted avg	0.80	0.79	0.79	17798
Accuracy: 0.7949769637037869				

Menggunakan Pustaka				
	precision	recall	f1-score	support
0	0.47	0.21	0.29	144
1	0.50	0.26	0.34	111
2	0.38	0.26	0.31	802
3	0.76	0.83	0.80	4047
4	0.62	0.79	0.70	2644
5	0.79	0.67	0.73	238
6	0.93	0.86	0.90	8542
7	0.79	0.77	0.78	1014
8	0.53	0.47	0.50	229
9	0.40	0.44	0.42	27
accuracy			0.80	17798
macro avg	0.62	0.56	0.58	17798
weighted avg	0.80	0.80	0.79	17798
Accuracy: 0.7953702663220586				



Model yang diimplementasikan secara mandiri menggunakan threading sebagai upaya untuk mempercepat perhitungan model. Penggunaan threading berguna dalam memparalelkan penemuan best split pada suatu node.

Lampiran

Pembagian Tugas

1. Dewantoro Triatmojo (13522011): Optimize Model KNN, Optimize Model ID3, Feature Engineering
2. Nyoman Ganadipa (13522066): Data Validation, Handling Missing Data, Handling Outlier Data, Remove Duplicates
3. Julian Chandra Sutadi (13522080): Model KNN, ID3, Naive Bayes
4. Rayhan Fadhlan Azka (13522095): Data Preprocessing, Save & Testing Model

Daftar Pustaka

Russell, Stuart, and Peter Norvig. (2021). Artificial Intelligence a Modern Approach 4th Edition. (Diakses pada 15 Desember 2024).

Mitchell, Tom. (2006). Generative and Discriminative Classifiers: Naive Bayes and Logistic Regression. Machine Learning. Carnegie Mellon University. (Diakses pada 15 Desember 2024).

Analytics Vidhya. (2022). Understanding Decision Trees [CART] in ML — Part I: Math & Intuition. Youtube. (Diakses pada 15 Desember 2024).

Analytics Vidhya. (2022). Understanding Decision Trees [CART] in ML — Part II: Building one from scratch. Youtube. (Diakses pada 15 Desember 2024).